High Level Synthesis of Globally Asynchronous Locally Synchronous Circuits

Christophe Wolinski, Mohammed Belhadj

HIGH LEVEL SYNTHESIS OF GLOBALLY ASYNCHRONOUS LOCALLY SYNCHRONOUS CIRCUITS

Krzysztof WOLINSKI and Mohammed BELHADJ

IRISA, Campus de Beaulieu
35042 Rennes, FRANCE

Abstract

This paper presents an approach for the design of *Globally Asynchronous Locally Synchronous* (GALS) circuits. The mixed style using asynchronous and synchronous circuits amalgamates the both styles best features. A language for high level specification of circuits is described. Then, the synthesis method that maps the algorithmic level specification in a net of GALS circuits is given. The asynchronous part is highlighted and avoidance of metastability is described. Finally, the link to existing CAD tools is given via VHDL.

1. Introduction

Advances in VLSI increese both area and speed of circuits. Easy handling of such circuits depends on the use of design automation tools (e.g. High, Register-Transfer and Logic level synthesis).

Synchronous automatic design tools are widespread and well-known as efficient methodologies. Correct functionalities of a synchronous system depend on the accuracy of the distribution of clock. Many attention in industry and academy has been given to the characteristics of the clock signals [4]. However, as the clock frequency increases, synchronous design becomes more difficult; problems like clock skew, metastabilty increase dramatically. A large part of ICs is devoted to clock generation and buffering.

A promising alternative is the use of asynchronous design where the absence of clock solves those problems, and offers good properties like composability and robustness [3]. But, asynchronous design have also their drawbacks: larger area, rarely mature industrial design tools, etc.

A good compromise seems to be the use of synchronous blocks that communicate by asynchronous techniques. For a discussion on advantages and disavantages of synchronous, asynchronous and mixed styles, see [5].

The work described here emphasizes the aspects of synthesis of Globally Asynchronous Locally Synchronous (GALS) circuits from the high level description language SIGNAL.

The originality of this approach is that the synthesis procedure is build upon the properties of the language. The asynchronous part is built with *delay-insensitive* elements [8]. This leads to robustness and composability of generated circuits.

The following section describes the input language SIGNAL, and its intermediate form. Then the synthesis method is described. A particular focus will be made in the asychronous part design. Then, a prototype using the Synopsys [1] VHDL synthesis environment is described.

2. The input language

SIGNAL is an equational language for the design of reactive applications [6]. It is a formally defined language with a small set of operators.

SIGNAL programs describe relationships between **signals** (a **signal** is a stream of typed values). Every signal possesses a *clock*[1] which determines if the signal is present or absent ($\perp$). The SIGNAL kernel is the minimum set of operators with which we can construct any SIGNAL program:

- The usual arithmetic and logic functions
- The \$ (delay) operator gives access to the last values of a signal.
- The under-sampling operator allows conditional extraction of values from a given signal:`Y := X when C`, Y is equal to X when the boolean signal C is TRUE.

X :	x_1	x_2	$\perp$	x_3	$\perp$	x_4	x_5	x_6	...
C :	F	T	F	F	T	T	$\perp$	T	...
Y :	$\perp$	x_2	$\perp$	$\perp$	$\perp$	x_4	$\perp$	x_6	...

- The default operator allows the deterministic merge of signals: `Z := X default Y`, Z merges X and Y with priority to X when both signals are present.

X :	x_1	x_2	$\perp$	x_3	$\perp$	...
Y :	y_1	$\perp$	y_2	$\perp$	y_3	...
Z :	x_1	x_2	y_2	x_3	y_3	...

Other operators have been defined using this kernel that permit the reduction of programming effort. For example `Y := X cell B` is the memory operator that can be coded using SIGNAL kernel operators[6].

A *Dynamic graph* (noted DG) is associated with a SIGNAL program. It describes the dependency of data and the

[1] *clock* is only a *logical* signal true when a signal is present and absent otherwise

relationship between *clocks*. The DG of SIGNAL programs are generated during the compilation process.

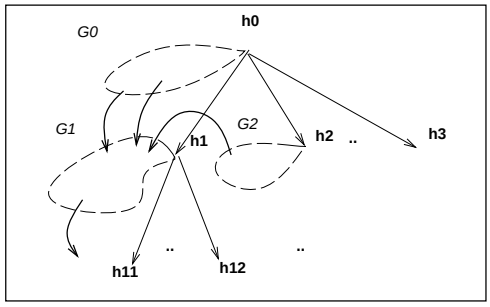

Fig. 1: SIGNAL internal graph representation

In Fig.1 h_0 is the fastest clock of the sub-system (in SIGNAL there is no general global clock, the fastest clock is computed for every system). The clocks h_1,h_2 and h_3 are sub-samplings of h_0. This means that if h_0 is absent then h_1,h_2 and h_3 are absent. This permits the reduction of computation frequency. A conditional data dependency graph is associated with every clock (e.g. G_0 with h_0). This graph represents all signals computed as frequently as the associated clock.

Let look at the following example to give an insight:

```
C:=(X1+X2 when (A>B)) default(X3*X4 when(A<=B))
```

If we suppose that the clock of `A` and `B` is h (Fig. 2), `A` value (a) is read when the clock h is true (idem. for `B`). The operation $x_1 + x_2$ is done only if h_1 is true (i.e. if A>B). Moreover, if h is absent we do not need to compute h_1 and h_2 and their corresponding graphs.

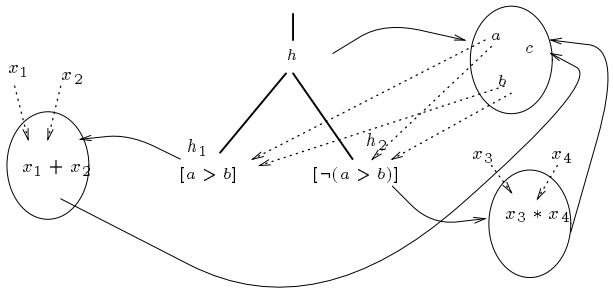

Fig. 2: An example of dynamic Graph

For a formal definition of *dynamic garphs*, see [6].

3. Synthesis method

This section presents how we transform the DG into a net of GALS circuits. By applying some transformations we produce a new graph (a net of processes that we can describe in SIGNAL). Ultimately a process on the net will correspond to an elementary processor (a GALS circuit).

A. Transformations

The synthesis process consists of two transformations:

- Construct a net composed of elements that implement the SIGNAL operators deterministic merge, subsampling, etc (we note the implemented operators as: c_or, c_cell, c_when, etc), fork[2] operators, and communication channels, using direct substitution from DG. This is *a direct implementation*.
- Partitioning the DG into subgraph containing at most two different clocks. Each subgraph will ultimately correspond to *an elemantary processor*.

Formally the two transformations corresponds to a closure of the graph. The resulting graph is a **net of elementary processors** and **fork** operators connected with **channels** synchronized by events (an event is the rising or falling edge of a signal).

Intuitively, every SIGNAL operator $\boxed{\text{op}}$ can be described as two operations, $\boxed{\text{op}}_v$ and $\boxed{\text{op}}_h$ corresponding to computation of value and *clock* of the output signal respectively.

$$c = a \boxed{\text{op}} b \equiv \begin{cases} c_v = a \boxed{\text{op}}_v b \\ c_h = a \boxed{\text{op}}_h b \end{cases} \quad (1)$$

where $a \equiv \{a_v, a_h\}$, a_v is the value of a and a_h represent its clock (true when a is presnt false otherwise). Note here that we have substitute the absence by the value false. To do so, we need a reference clock: the fastest clock of the net. *Clock* here refers simply to a sequence of edge triggered asynchronous "events" and not to a physical synchronous clock.

Signals in SIGNAL language are replaced by event-synchronized channels (hand-shake). A signal C, is defined as follows : $\{c_v, c_{hv}, c_h, c_{hack}\}$ where c_v is the value of the signal C in terms of the SIGNAL language, c_{hv} is the value of the clock of signal C (i.e. if c_{hv} is true than C is present for the current instant of reference clock c_h, otherwise it is absent), c_h represents the reference clock or the fastest clock (for this part of the net). It is represented physically by an event. c_{hack} is an acknowledgment that corresponds to the end of possible computation for the current tick of c_h.

So, the opertors are replaced as described in (1), adding a local conditioning mechanism for the $\boxed{\text{op}}_v$ and $\boxed{\text{op}}_h$ computation, and adding a mechanism for the output reference *clock* computation. The signals are replaced by channels. Finally, the substituted graph is partitioned.

B. Resulting Net

After the two transformations we obtain a net of processes (Fig. 3), where data and clock transfer use hand-shake.

The synthesized subgraph (a process or physically a processor) is composed of an asynchronous control part and a synchronous part for the computation of $\boxed{\text{op}}_v$ and

[2] fork broadcasts its input signal to a number of outputs

$\boxed{\text{op}}_h$. The asynchronous part ensures that the computations are done when necessary . The decision is made dynamically (and locally), by considering the values of clock of input channels with respect to the reference clock and the position of the element in the net.

For (Fig. 2) example, the asynchronous part of processors P1 and P2 (Fig. 3) enable computation in synchronous parts if a_{hv}, a_h, a_v arrive, and a_{hv} is true. Moreover, $(a > b)$ from P1 and $(a \leq b)$ must be true. If the computation is not necessary and the result of the operation is needed for another computation, we send in the output channel the clock value (hv) false (the value of channel is not important). Note, that the number of request and acknowledge signals are reduced (e.g signals from the same synchronous part of a processor use one request).

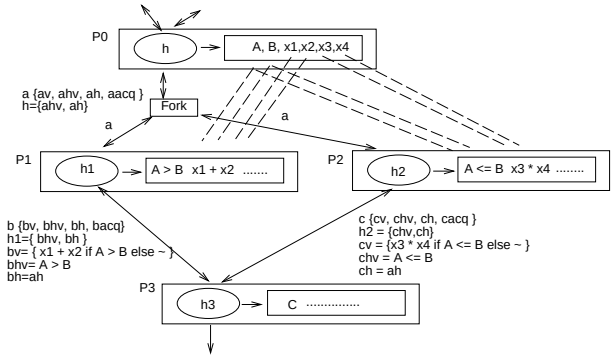

Fig. 3: A transformed Graph

C. Elementary processor

An elementary processor (Fig.4) is composed of: input channels and output channels and perhaps, a parallel interface using "hand-shake" that permit the implementation of synchronous procedure call, by sending parameters and receiving results.

An elementary processor is composed of two parts: a synchronous part and an asynchronous one. The asynchronous part (implemented with *delay-insensitive* elements: *select, c-muller, merge, etc* [8]) determines when the processor is supposed to compute its results. This decision is taken dynamically. The synchronous part executes the computation corresponding to the reduced part of the net.

The synchronous part can be optimized using classical methods (FSM reduction, hardware sharing, etc).

4. Asynchronous part

The asynchronous part (see Fig. 4) of an elementary processor enables the execution of operators $\boxed{\text{op}}_a$ and $\boxed{\text{op}}_h$ corresponding to the computation (associated with this processor), evaluates the reference *clock* (signals CH, CHV), and generates acknowledgments (Aacq, Bacq).

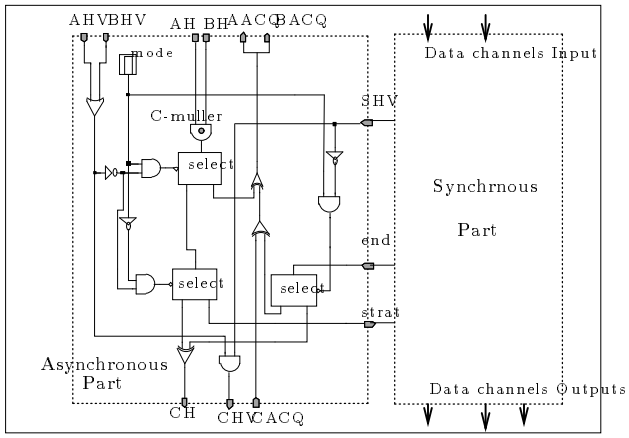

Fig. 4: Elementary processor

The decision of enabling the computation takes into account Ahv, Bhv and position of the processor in the net(if the processor outputs are not used as input in other processors optimization on computation frequency is possible).

The value SHV (Fig. 4) is given by the synchronous part. It corresponds to the current value of the *clock*. The synchronous part is awoken when an event occur in the signal **start**. There is an end of the computation (an event occurs on **end**) either when SHV is false or all computations corresponding to the synchronous part are finished.

A. Hand-shaking problems

Our implementation uses a two phases protocol [8]. The generated architecture must guarantee that a data arrives before its corresponding request signal. Data used by synchronous part are prepared by preceding processors in the net (e.g. Fig. 5).

The events AH and BH are generated by asynchronous parts of processors A and B, after their respective synchronous processors ended their computations. Then, the data must be stable before the asynchronous part of the processor C generates the computation event.

Without the time corresponding to connections routing delays, the request events (e.g AH, BH) have a delay:

$$\Delta T = 2T_{osc} + 3T_{select} + T_{xor} + T_{c-muller}$$

regarding the end of computation of data (T_{osc}: delay of 1 cycle of physical clock oscillator, T_{select}: delay of a select operator ...).

The decision taken by the asynchronous part uses the hand-shake signals AH, BH and the logical signals AHV and BHV. The processor C operates correctly if AHV and BHV are stable before the arrival of the events AH and BH (request). Any asynchronous part ensures the correct behavior (of the hand-shake) because it generates the request signal H (e.g AH,BH,CH) after the signal HV is stable.

If the routing conditions are not arbitrary, the hand-shaking operates correctly.

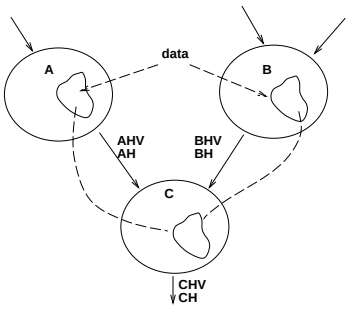

Fig. 5: A part of a net

B. Metastability problems

The generated architecture being composed of asynchronous and synchronous parts, the question of metastability may arize. The asynchronous part uses *delay-insensitive* operators but the synchronous part uses normal flip-flop. Previous works [5][7] use special flip-flop called *Q-modules* to handle the interface between synchronous modules and hand-shake circuits. In the following we describe how the metastability can be avoided in our case.

There are two possible cases for the generation of CHV and CH signals:

- No computation is needed: the delay for the computation of CHV (2) is smaller than the delay needed to produce the event CH (3).

$$\Delta T_{AHtoCHV} = T_{or} + T_{and}(2)$$

$$\Delta T_{AHtoCH} = T_{c-muller} + 2T_{select} + T_{xor}(3)$$

- Computation is necessary: when the computation ends the SHV event is stable and CH is produced afterwards (4), at the time of the request of the synchronous part of elementary process signal **end** (5).

$$\Delta T_{SHVtoCHV} = T_{and}(4)$$

$$\Delta T_{\mathbf{end}toCH} = T_{osc} + T_{select} + T_{xor}(5)$$

The synchronization between asynchronous and synchronous parts is done by a synchronous automaton that samples the signal **start**. To avoid metastability (or more accurately to minimize it) the sampling is done in the falling edge of the internal physical clock, while the automaton is activated on the rising edge.

5. Experimental results

We have synthesized an architecture from a SIGNAL program describing a control process (see [9] for a complete example). The result was described in structural VHDL and was validated under the VHDL simulation environment.

Synchronous parts were automatically synthesized by Synopsys [1], while asynchronous parts were generated separately. An implementation in the Synopsys and Xilinx FPGA [2] environments has been achieved (Fig. 6).

6. Conclusion and further works

We propose a general method for the synthesis of GALS circuits from SIGNAL specification. The synthesis procedure transforms the intermediate form into another graph representing the implementation in terms of circuits behavior. The synthesis uses the notion of *local clock* in SIGNAL which reduces the frequency of computation.

The main advantages are: absence of global clock (no skew problem), dynamic optimization, and access to VHDL synthesis tools and SIGNAL environment (offering possibility for formal proof, simulation, etc). The drawback of this approach is the size of the resulting circuit, we are currently working on the optimization of the synthesis results.

The complete automatic translation from SIGNAL to Xilinx FPGA is under development, and in near future a cell generator for the asynchronous part (in CMOS technology) will be developed. Moreover, a study is conducted separately for generating distributed code for parallel machines using the same partitioning of SIGNAL programs, to permit Hardware/Software codesign.

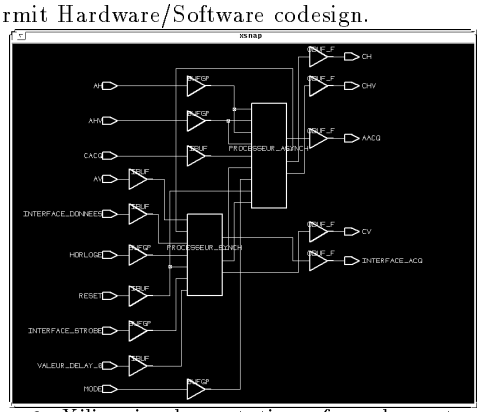

Fig. 6: Xilinx implementation of an elementary processor

References

[1] *Synopsys VHDL Compiler Reference Manual.*

[2] *The XC4000 Data Book, Programmable Gate array.* XILINX, Inc. 1990.

[3] Special issue on Asynchronous systems. *Integration, the VLSI journal*, 15, 1993.

[4] E. Freidman. Clock distribution design in VLSI circuits- an overview. In *IEEE International Symposium on Circuits and Systems,*, May 1993.

[5] G. Goplalakrishnan and L. Josephson. Towards amalgamating the synchronous and asynchronous styles. In *TAU'93, ACM Workshop on Timing Issues in the specification and synthesis of digital systems*, 1993.

[6] P. Le Guernic, Th. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, sep 1991.

[7] F. Rosenberger, Ch. Molnar, Th. Chaney and T. Fang. Q-modules: internally clocked delay insensitive modules. *IEEE Trans. on Comp.*, 37(9):1005–1018, Sep 1988.

[8] I. E. Sutherland. Micropipelines. *Communication of ACM*, 32(6):720–738, jun 1989.

[9] K. WOLINSKI and M. BELHADJ. *Vers la synthèse automatique de programmes SIGNAL.* Technical Report 746, IRISA, 1993.(In French)